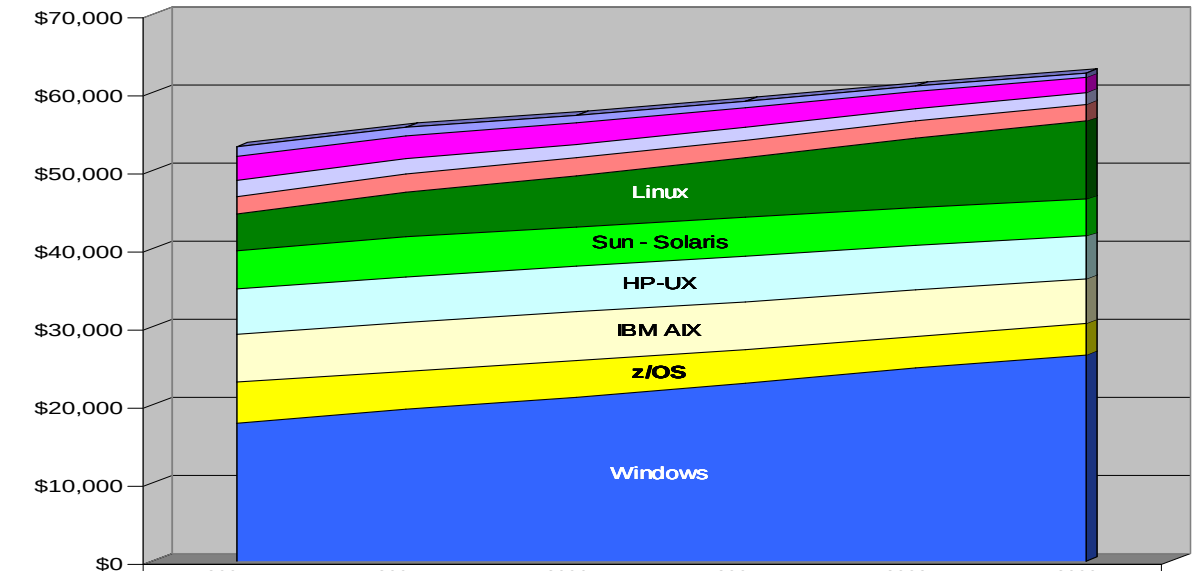
Platform Solutions, Inc.

Itanium as a Horizontal Microcode Engine for Legacy Architecture Enablement

Ron Hilton,
PSI Founder & CTO

WW Server Market Revenue



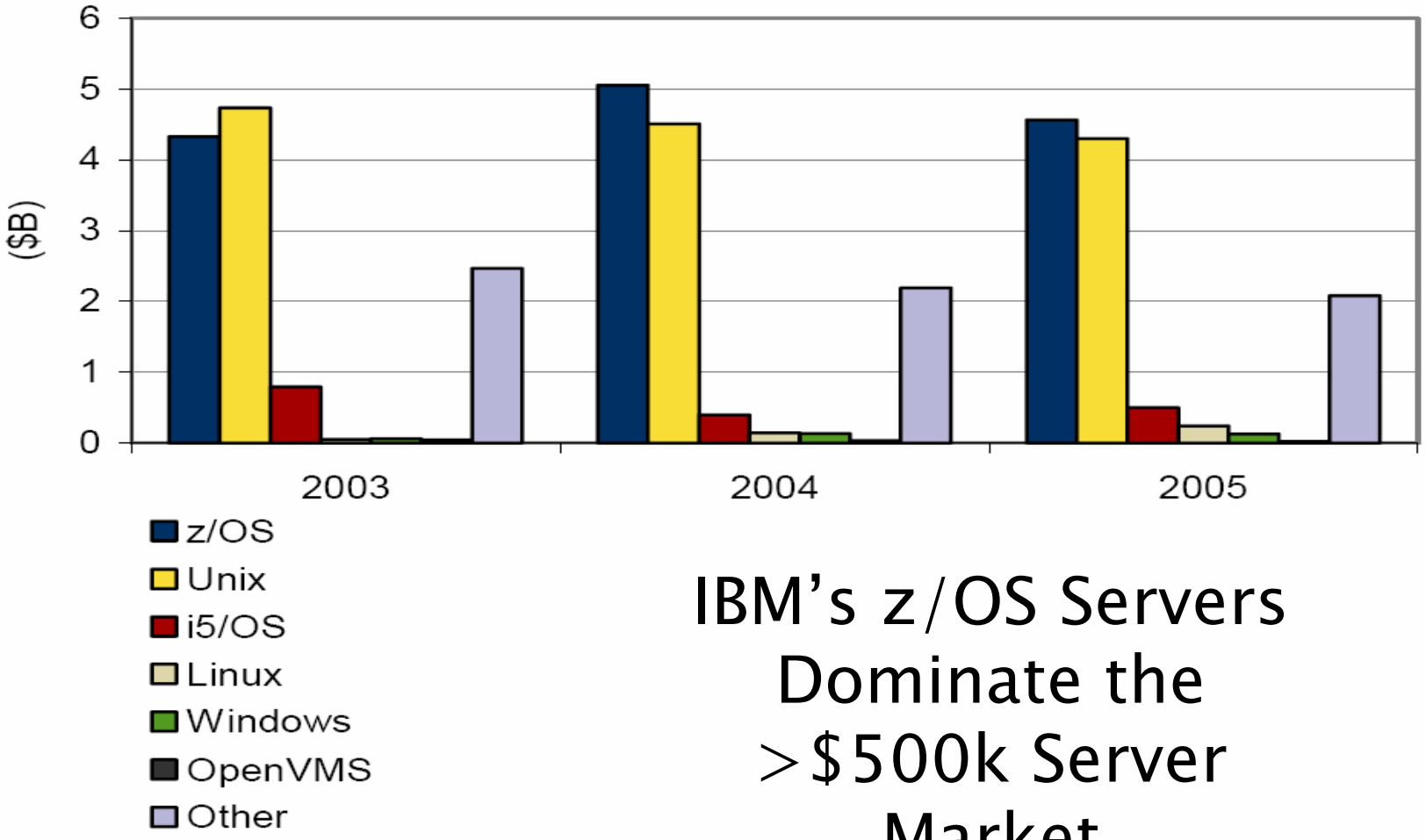
	2004	2005	2006	2007	2008	2009
NetWare	$1,222	$1,067	$967	$862	$754	$614
Other	$3,101	$2,979	$2,754	$2,503	$2,205	$1,946
OS/400	$2,093	$1,844	$1,692	$1,606	$1,546	$1,464
All other Unix	$2,290	$2,351	$2,336	$2,308	$2,252	$2,173
Linux	$4,617	$5,775	$6,483	$7,626	$8,839	$10,002
Sun - Solaris	$4,963	$5,093	$5,062	$5,002	$4,880	$4,708
HP-UX	$5,726	$5,876	$5,841	$5,771	$5,630	$5,432
IBM AIX	$6,108	$6,268	$6,230	$6,156	$6,006	$5,794
OS/390	$5,380	$4,799	$4,658	$4,264	$4,040	$3,933
Windows	$17,576	$19,456	$21,027	$22,766	$24,695	$26,441

- z/OS is approximately a $5B annual market for Hardware
- Similar is size to HP-UX, Solaris, and AIX markets
- IBM is currently the only vendor with z/OS compatible HW after Hitachi and Amdahl exited market in 2000
- ~15,000 Units installed world wide
- Machine are replaced and upgraded with capacity every 3-4 years

Source: IDC

$500k & Above Server Market

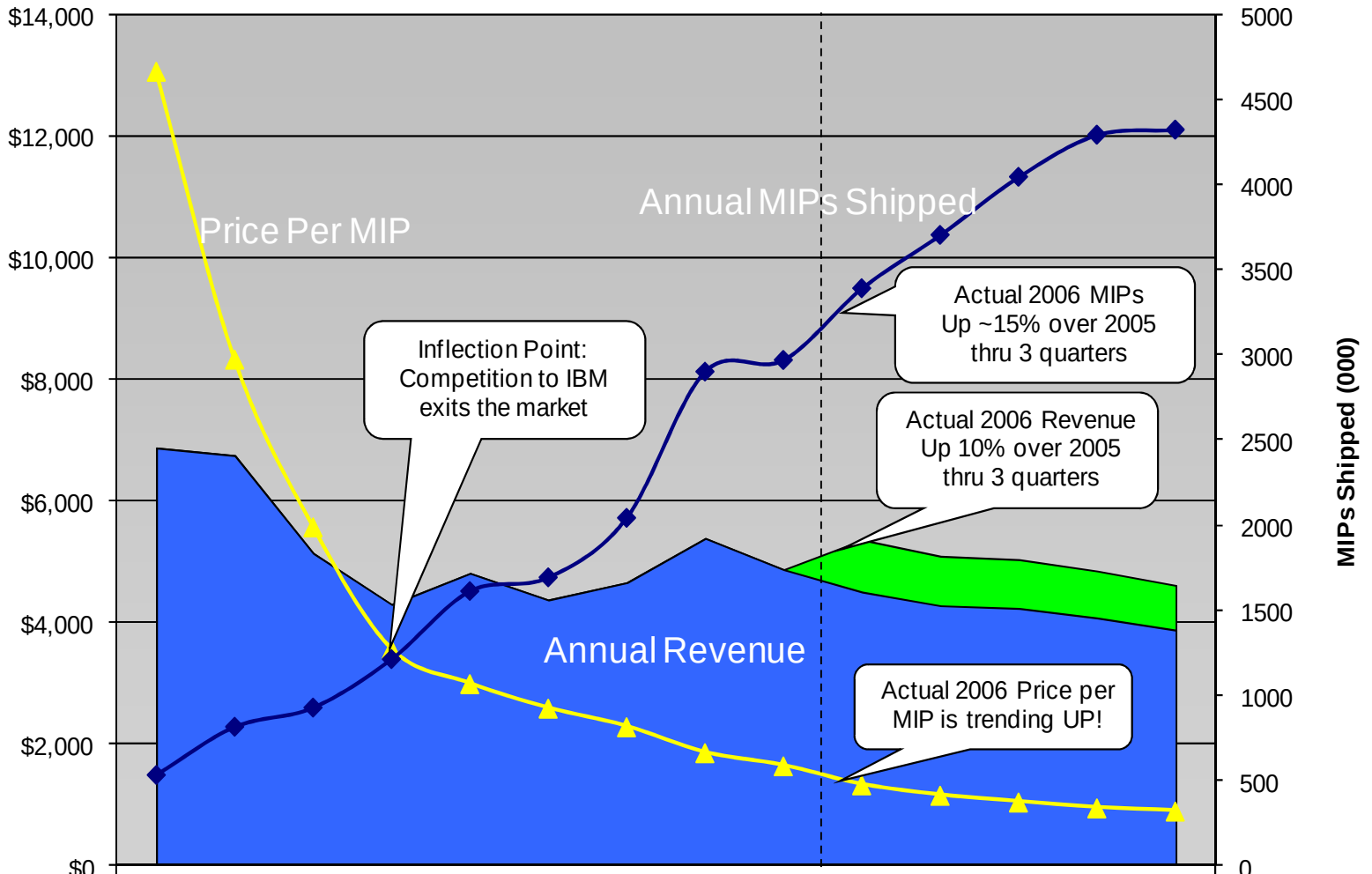
Worldwide High-End Enterprise Server Factory Revenue by Operating System, 2003–2005



IBM's z/OS Servers Dominate the >$500k Server Market

Source: IDC, 2006

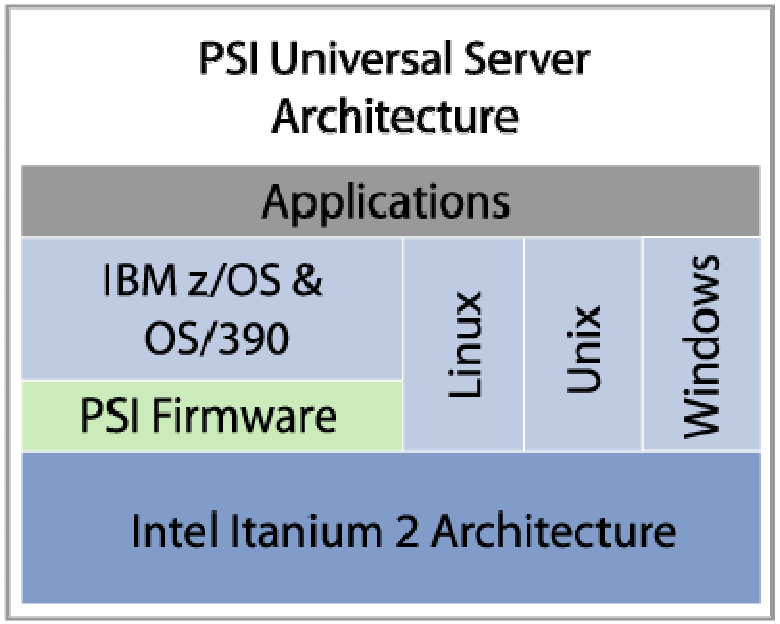
zOS Market Price Performance

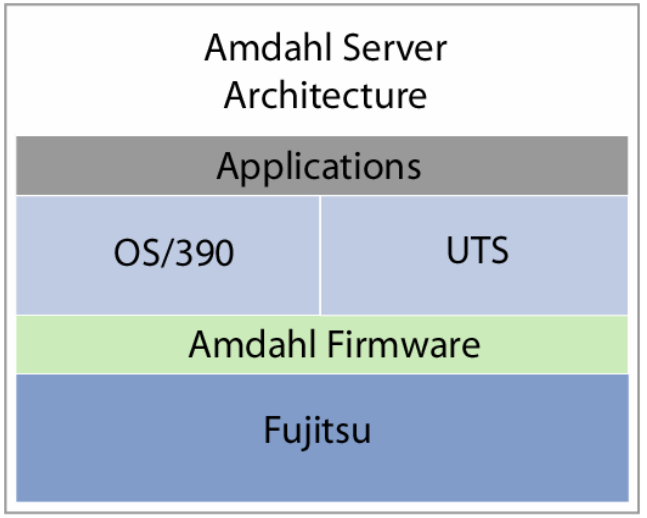


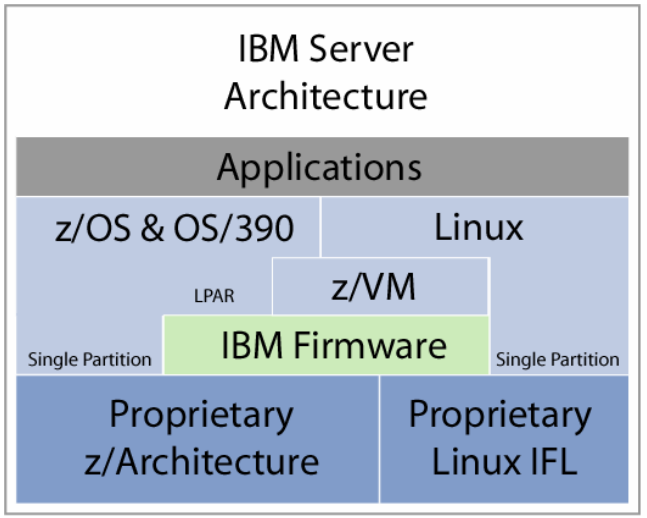
	1997	1998	1999	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010
Actual Adjusted Rev. Trend	6865	6739	5138	4297	4809	4370	4656	5381	4868	5354.8	5087.1	5032.4	4845.7	4609.0
Revenue ($M)	6865	6739	5138	4297	4809	4370	4656	5381	4868	4502	4277	4231	4074	3875
Price per MIP ($M)	13051.	8319.7	5566.6	3557.1	2988.8	2585.8	2282.3	1854.8	1639.0	1327.2	1154.0	1045.2	948.32	895.74
MIPS Shipped (000)	526	810	923	1208	1609	1690	2040	2901	2970	3392	3706	4048	4296	4326

Source: IDC, Merrill Lynch, PSI Analysis

Architecture Comparison







PSI Historical Background - Amdahl

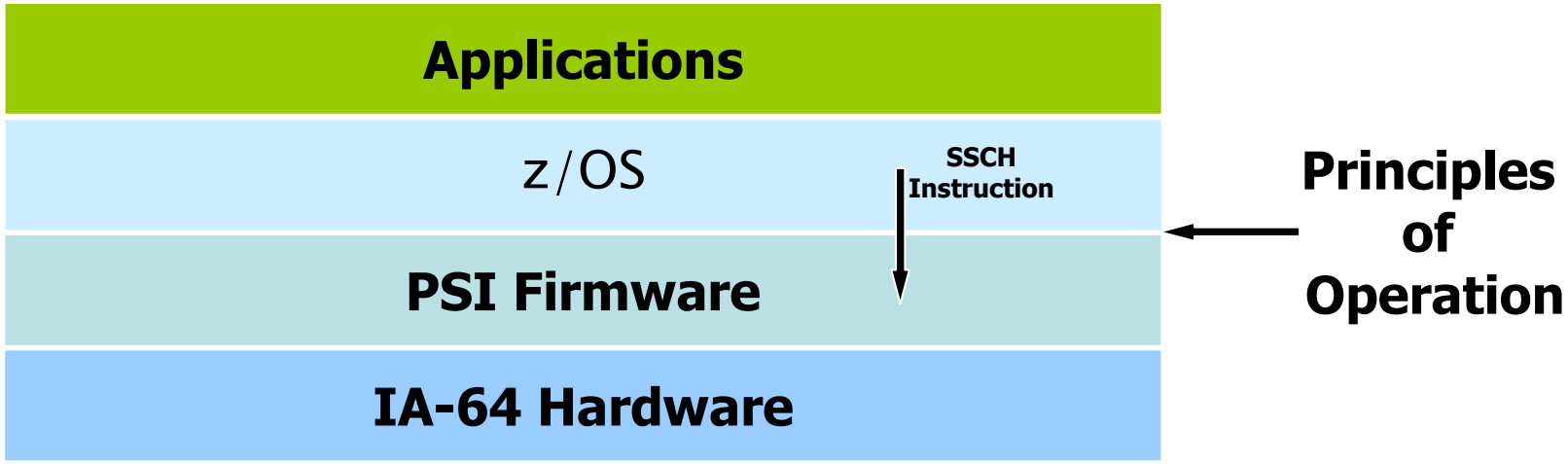
- Amdahl invented Fast Assist Mode (FAM) as a means of rapidly responding to IBM architectural changes through emulation in late ’70s
- Partitioning was another Amdahl first – Multiple Domain Facility (MDF) evolved from FAM hypervisor in early ’80s
- Amdahl Hardware Simulator (AHSIM) used simulation technology to facilitate development of future Amdahl hardware/software in mid ’80s
- Experimental “Firebird” and “SoftIron” S/390-on-IA32 emulators developed in late ’80s
- Fundamental Software, Inc. (FSI) founded in ’91 by ex-Amdahl employees to commercialize S/390-on-IA32

PSI Historical Background – Amdahl & Intel

- Intel and Amdahl (codename ‘Vail’) began collaborating on S/390-on-IA64 technology in ’95
- Initial studies involved small test cases translated manually into IA64 code
- Developed initial “Manta” prototype on PPC platform to test basic virtualization concepts in ’96
- “Merlin” follow-on to AHSIM developed from SoftIron/Manta base
- Developed “Stingray” prototype on Intel IA64 “Gambit” simulator, significantly enhancing Manta algorithms in ’97
- Platform Solutions, Inc. (PSI) was spun off from Amdahl in Jan ’99 to commercialize S/390-on-IA64 using Stingray/Merlin base

Why PSI JIT Virtualization works with z/OS

- PSI is strictly instruction-level virtualization
- There is no emulation at the API level
- I/O and the Hardware System Area (HSA) are only accessible through architecturally-defined instructions



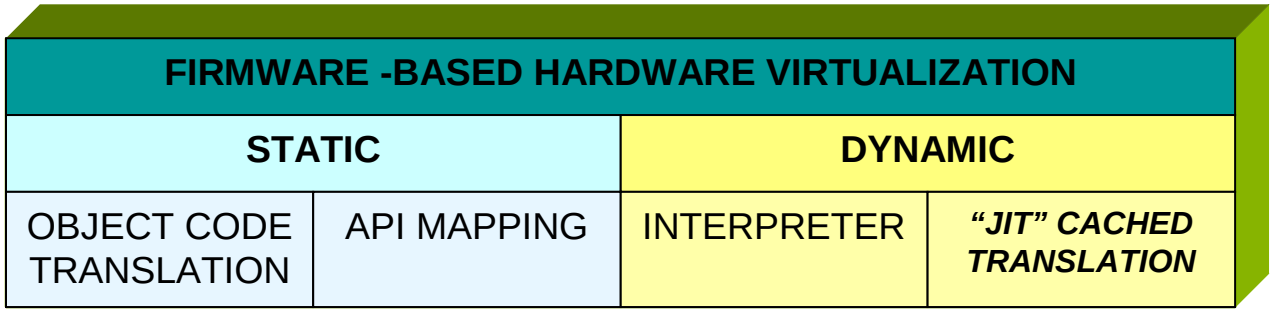
Mainframe Legacy Architecture Requirements

- Applications are portable across ALL Mainframe platforms since 1960's
  - S/360 – S/370 – 370/XA – ESA/390 – z/Architecture
- Primary compatibility specification is the IBM Principles of Operation
  - Most recent version is 1150 pages
  - Contains detailed descriptions of instructions and facilities
  - Limits to model-dependent behaviors are described
- OS is not tied to a specific machine
  - Architectural level sets define machine/OS compatibility
  - z/OS simulates newer features when appropriate
  - z/OS performs extensive feature checks on startup

Diagnostics Overview

- PSI has licensed Amdahl's diagnostics (BUPs and SLTs)
- Bring-up Programs (BUPs)
  - Small (generally <2000 lines of code)
  - Each BUP tests one or two instructions or a facility
  - Go/no-go tests with no user interface
- System-Level Tests (SLTs)
  - SLTs are essentially stand-alone diagnostic operating systems
  - Random code sequences are generated, executed and checked for correctness
  - Alpha and 8E7 are primarily CPU diagnostics
  - DIRT and HOT primarily test I/O

Virtualization Technology – Basic Approaches

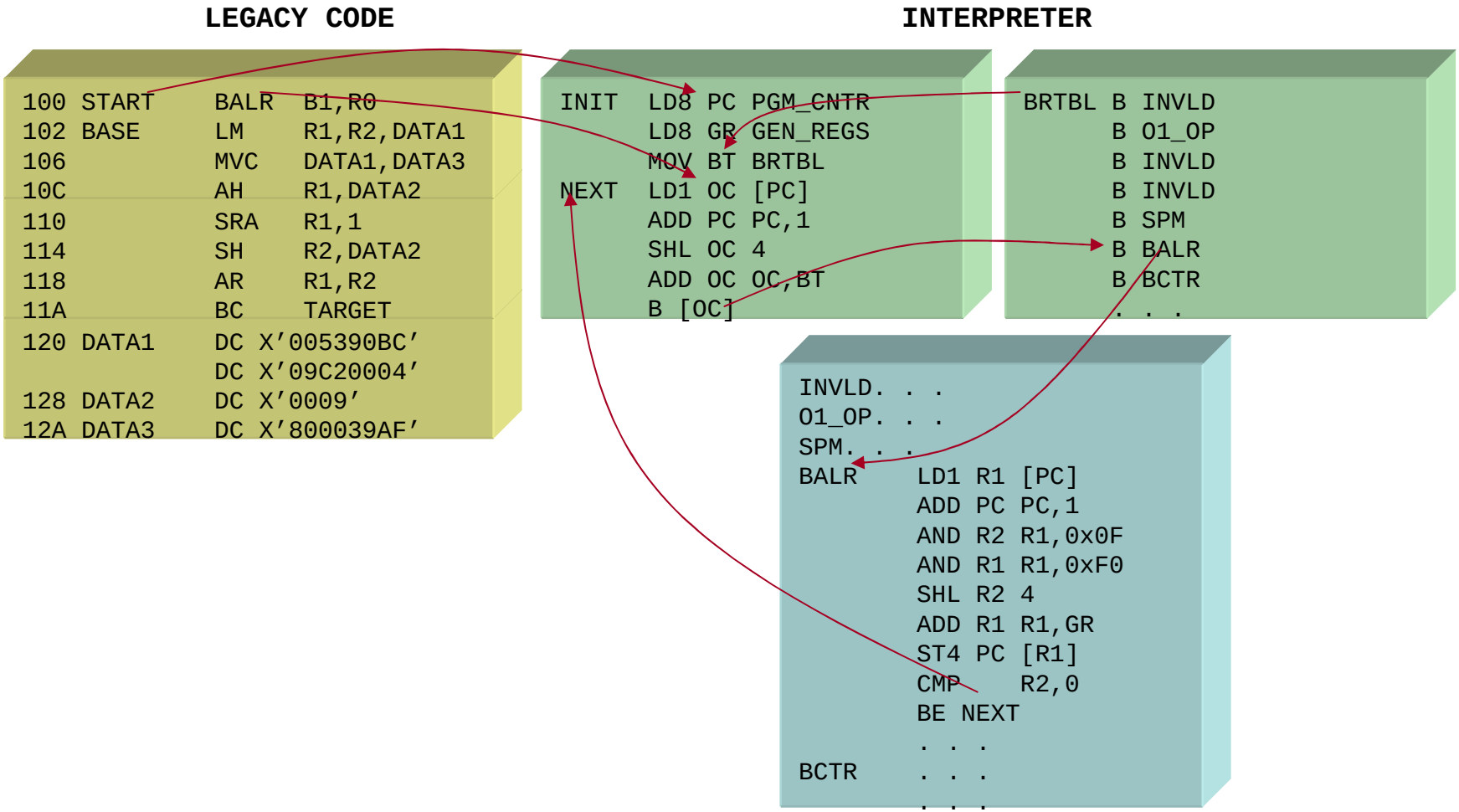


- Static – Translated Code (prior to run-time):
  - Object Code Translation – 'compiled' with legacy binary as 'source'.
  - API mapping – Legacy OS functions mapped to native OS calls.
- Dynamic – Execution Code (during run-time):
  - Interpreter – Each legacy instruction fetched, parsed, and executed.
  - ***JIT - Piecemeal object code translation and caching "on the fly."***

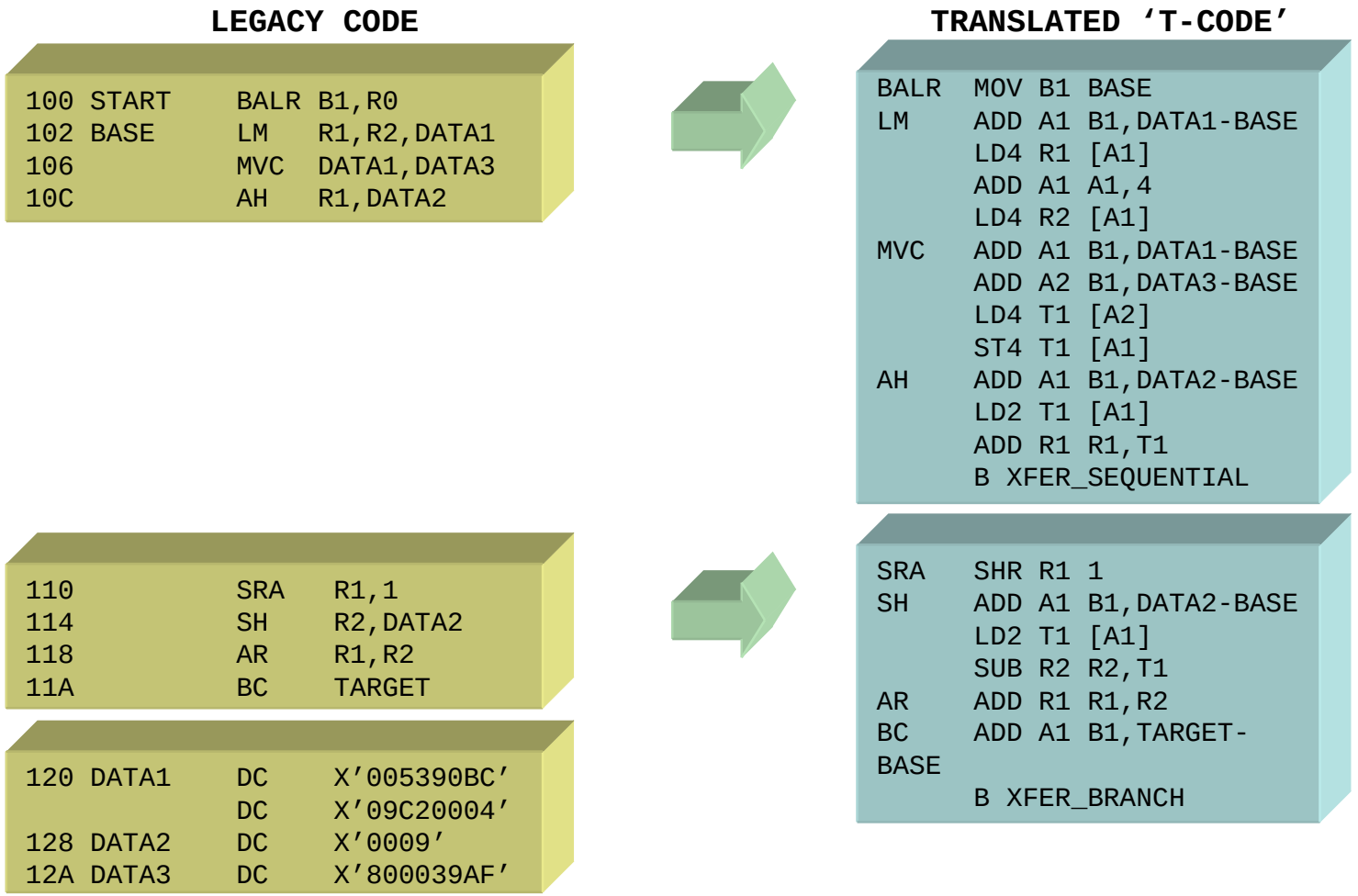
Virtualization Technology - Basic Issues

- Static Translation Issues
  - Indirect branches predominate
  - Variable-length instructions
  - Not user-transparent
- API Translation Issues
  - MVS has no crisp “API” as such
  - OS translation miss rate already low
- Interpretation Issues
  - Comparatively slow
- JIT Translation Issues
  - Self-modifying code
  - Run-time translation overhead – CPU utilization is already high
  - Translating ahead is problematic – same issues as static translation

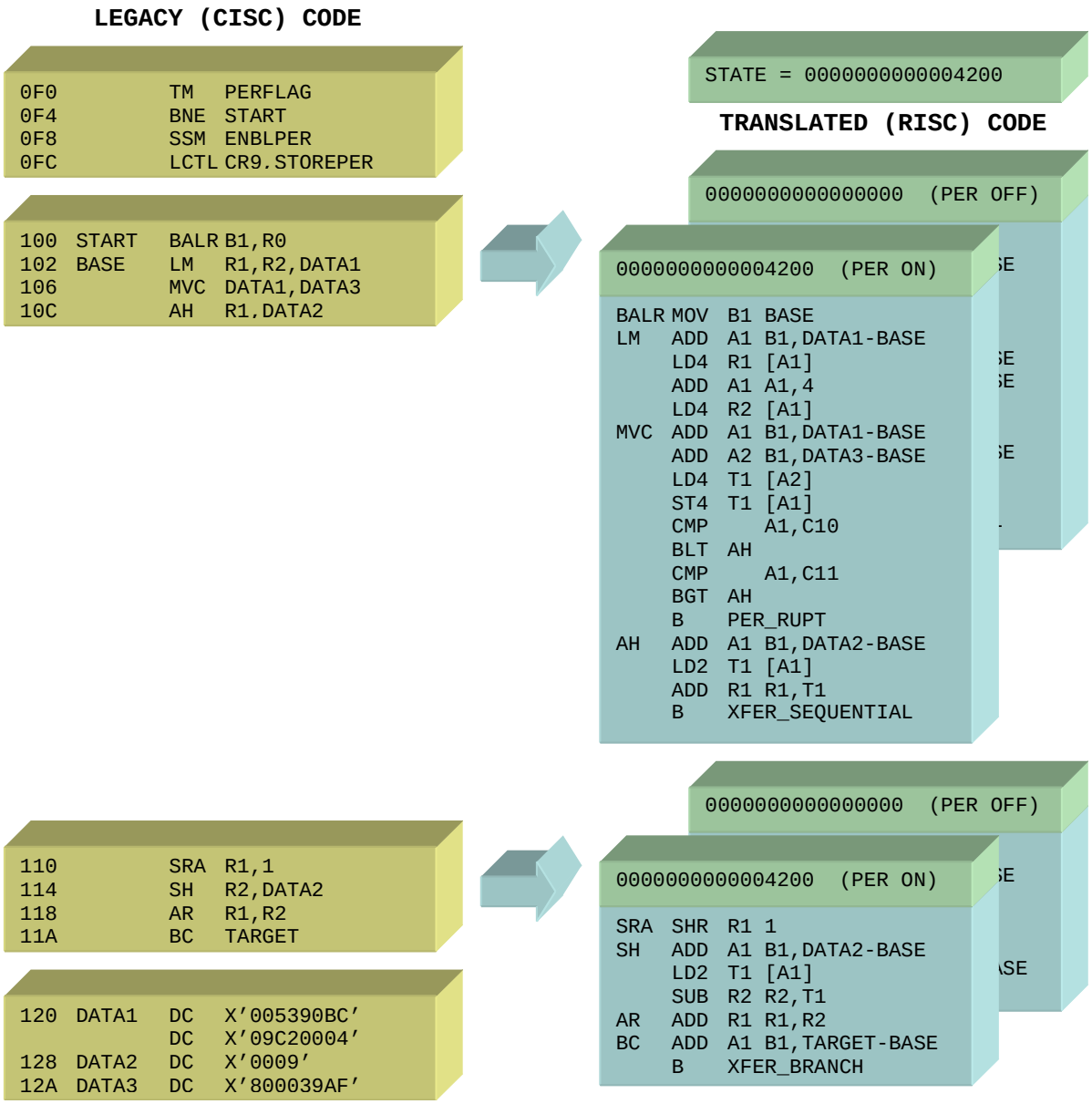
Interpretation – Low-performing Approach



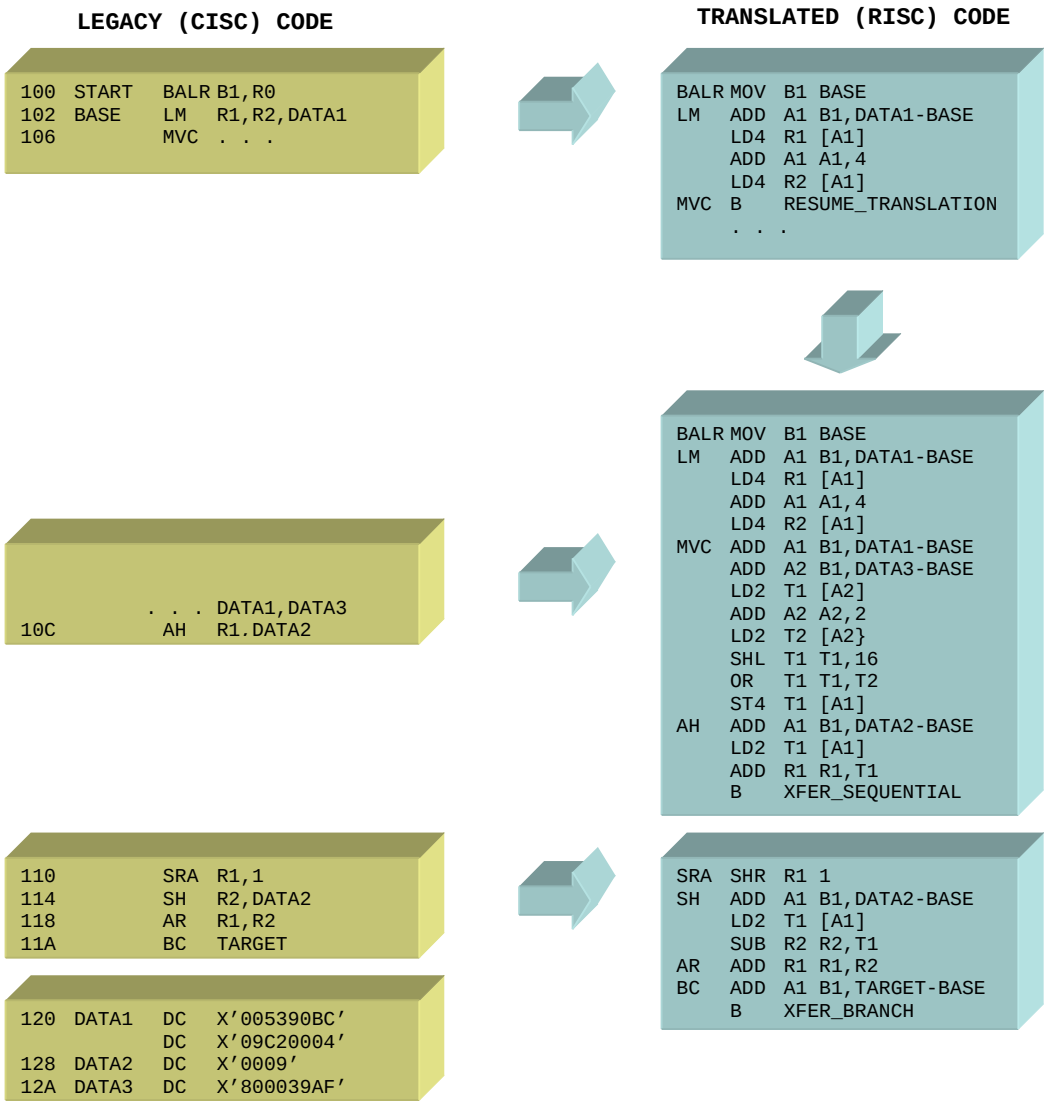
PSI High Performing - JIT Binary Translation



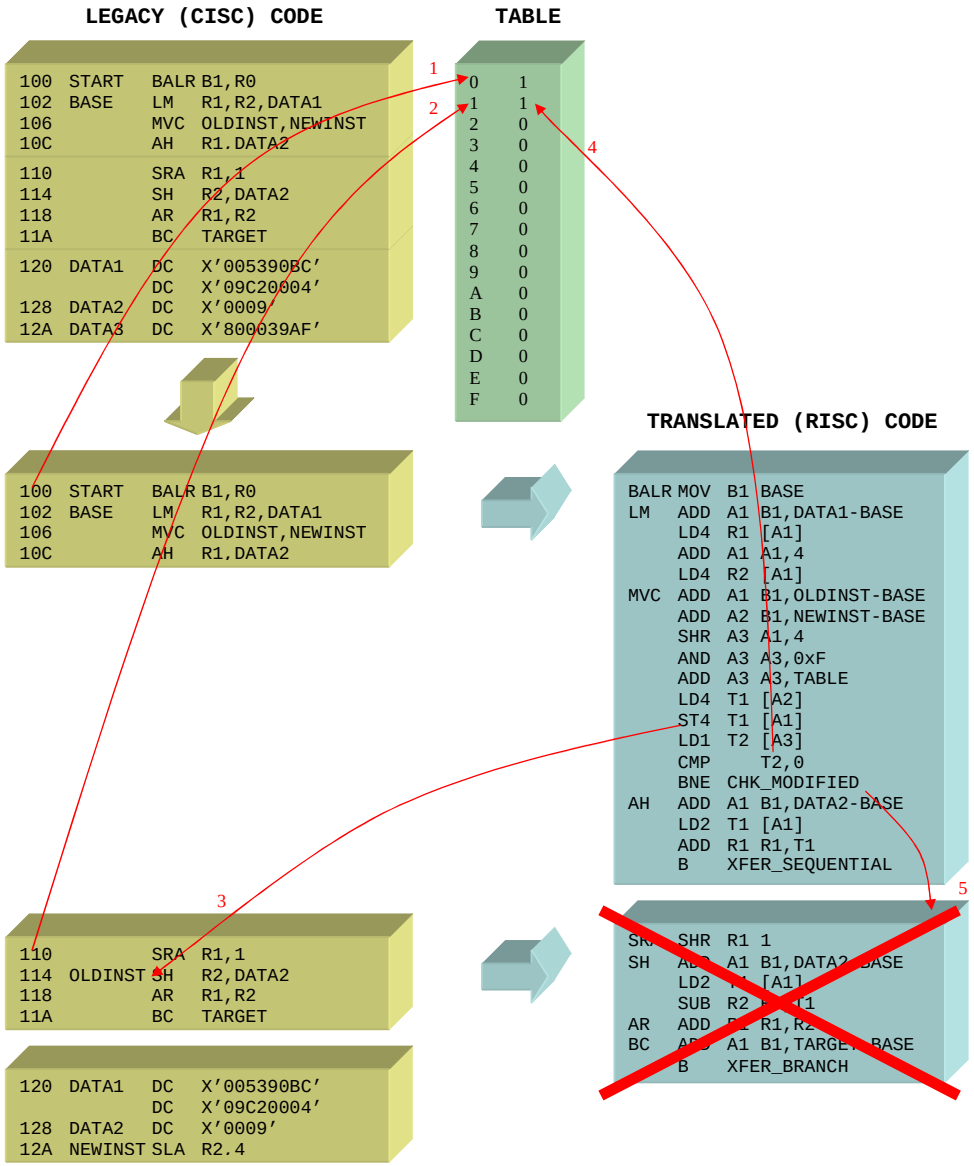
JIT Translation – State-specific Variants



JIT Translation - Memory Address Prediction

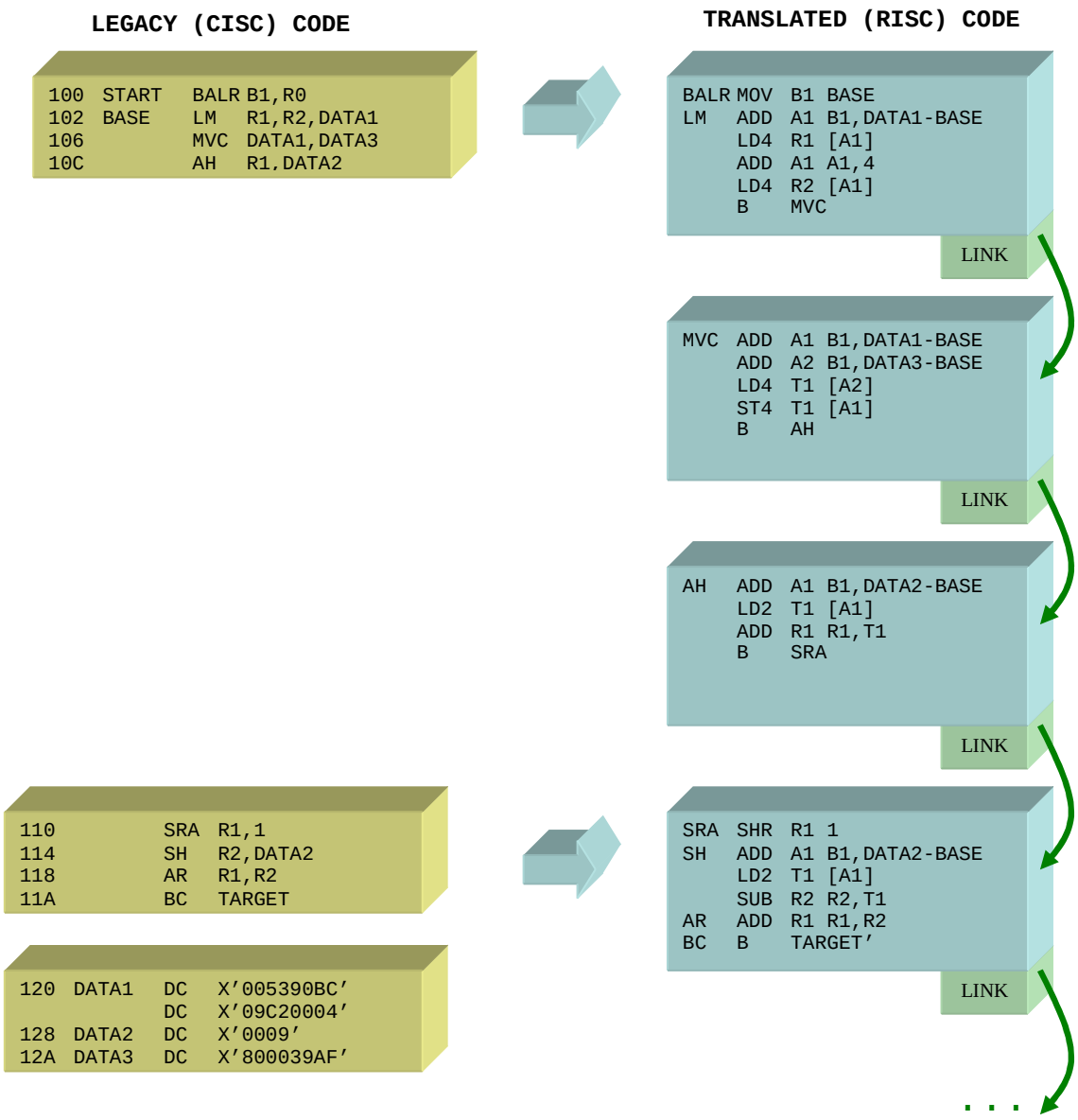


JIT Translation - Self-modifying Code



Platform Solutions, Inc.

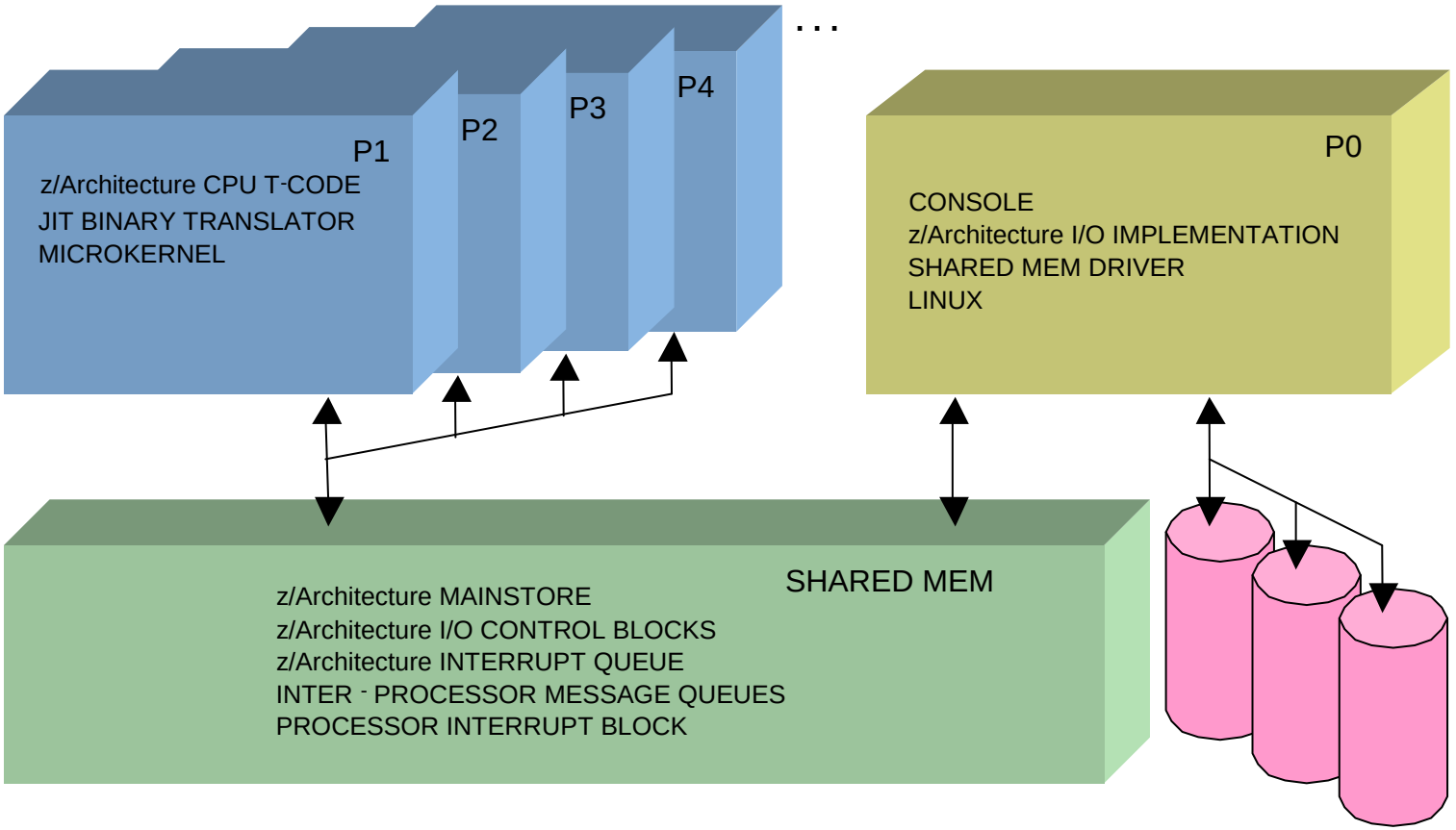
JIT Translation – Flexible Caching



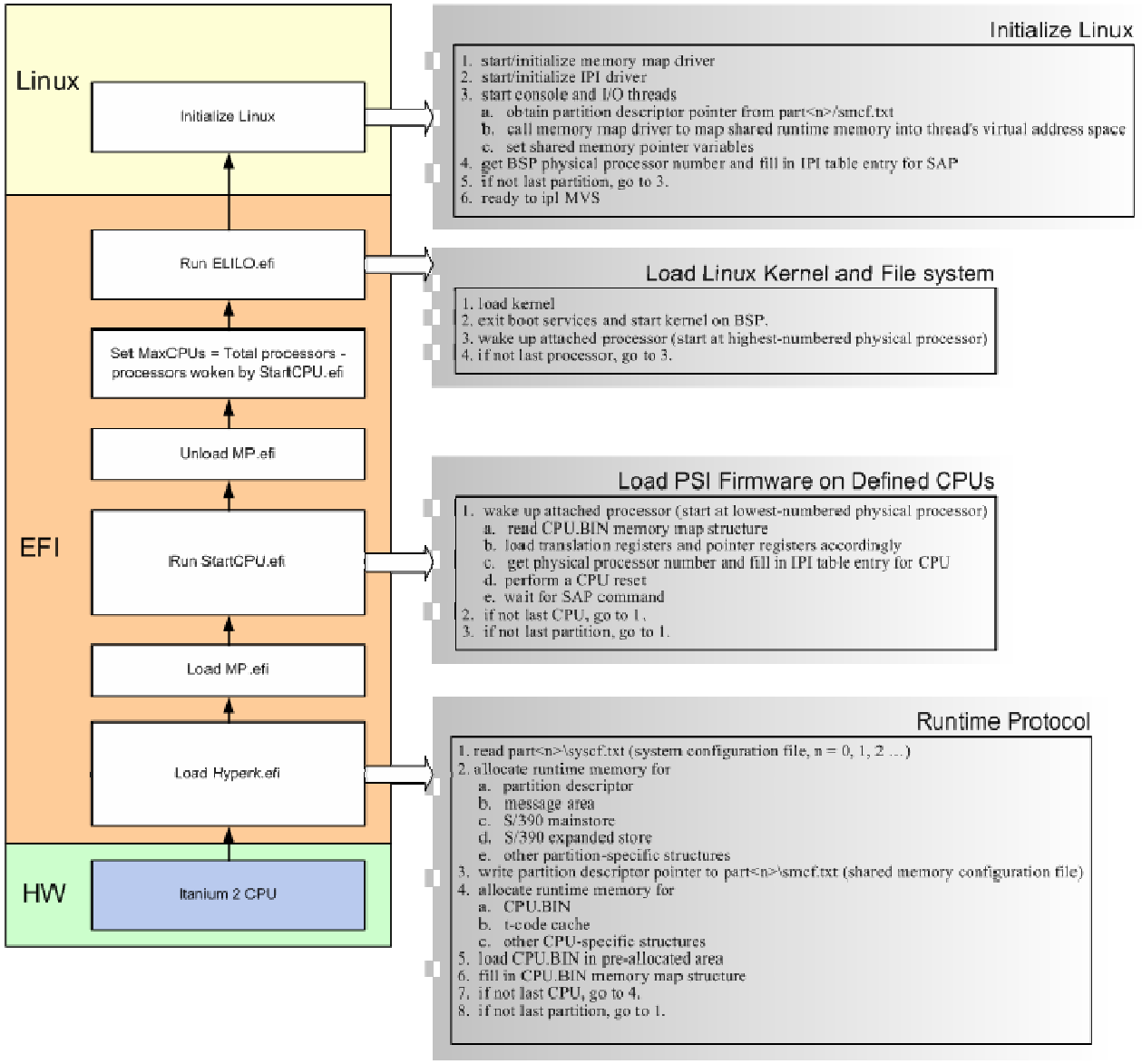
JIT Translation – Direct Branching

- Branch base register tends to be static
- If branch base register is unmodified, target stream is queued for translation
- Translation continues until all queued streams are exhausted
- T-cache entries that share a common base register are logically grouped together
- Branch base register contents are checked whenever a logical group is entered

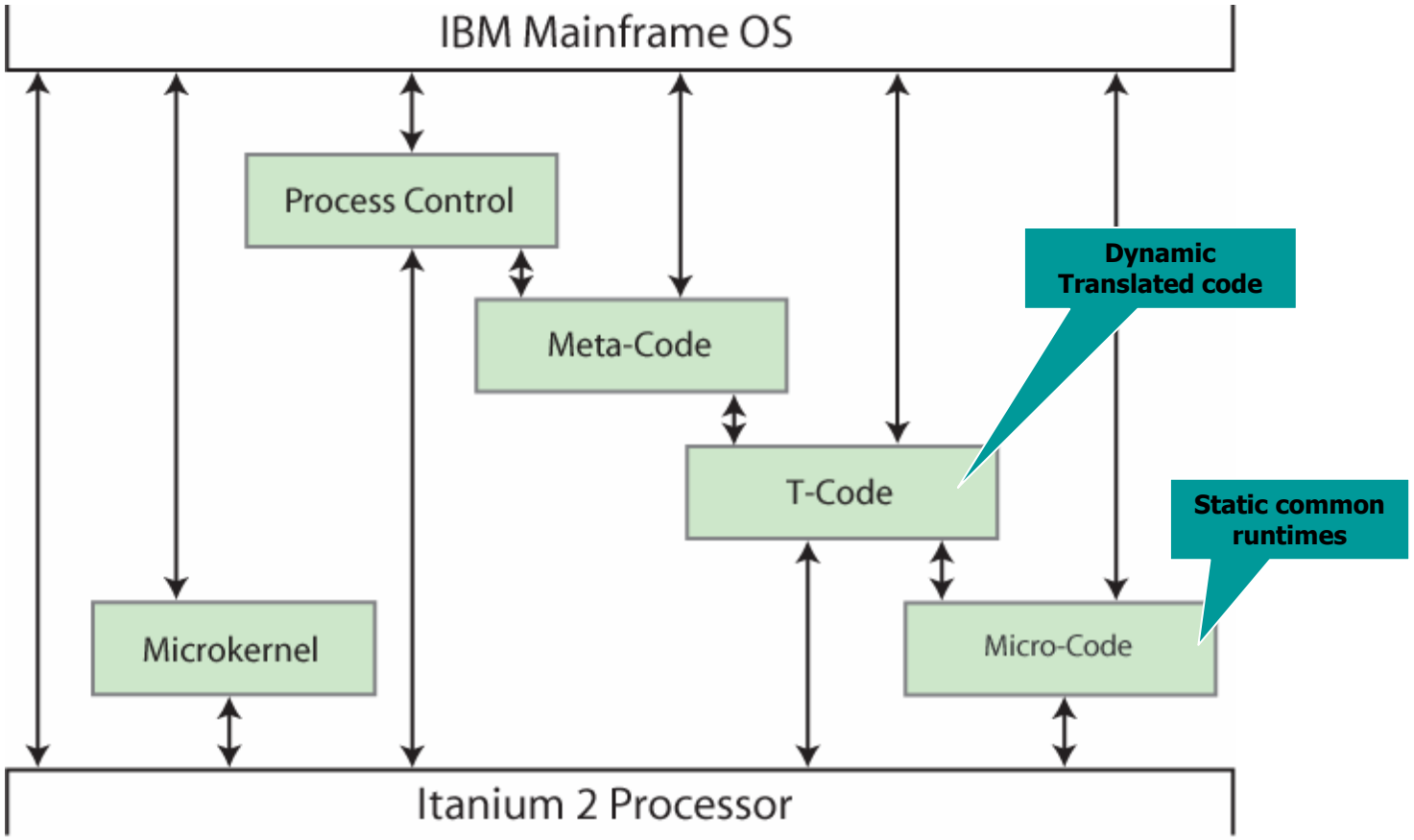
PSI System Block Diagram



PSI Firmware Load Sequence

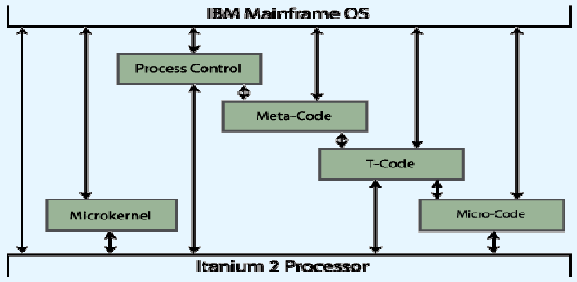


PSI Processor Technology



PSI technology loads at EFI load time and is not dependent on any other operating system software

PSI Heterogeneous System Architecture

Mainframe Applications	L/Unix Apps	WIN Apps
IBM OS/390 & z/OS Operating System	Unix and System services Unix/Linux Enablement (ISV)	WIN 2003 and system services Device and communication management
 I/O Services System Services Layer		
EFI (Initializer)		
Hardware Layer (Intel) IA 64		
Bus, Channel, High Speed External Interfaces		
Storage Layers		

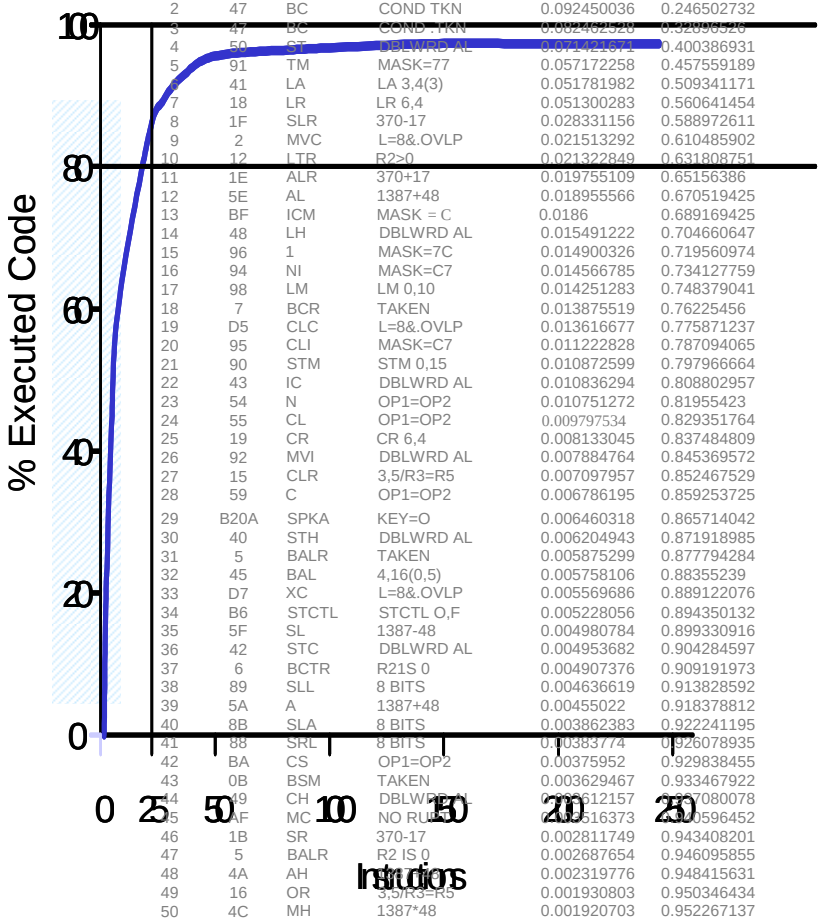
… A common enterprise platform for all software asset management, processing, and storage management

Instruction Distribution is the Key

- **Performance tuning of the "Primal Instruction Set"**

- 10% of the instructions account for 80% of those executed.

- Tuning "primal set" of instructions allows exploitation host machine at native speeds.



INSTR COUNT	OPCD	MNEM	NOTES	TSO FREQ	CUML FREQ
1	58	L	DBLWRD AL	0.154052697	0.154052697
2	47	BC	COND TKN	0.092450036	0.246502732
3	47	BC	COND TKN	0.082462328	0.328965328
4	50	ST	DBLWRD AL	0.071421871	0.400386931
5	91	TM	MASK=77	0.057172258	0.457559189
6	41	LA	LA 3,4(3)	0.051781982	0.509341171
7	18	LR	LR 6,4	0.051300283	0.560641454
8	1F	SLR	370-17	0.028331156	0.588972611
9	2	MVC	L=8&.OVLP	0.021513292	0.610485902
10	12	LTR	R2>0	0.021322849	0.631808751
11	1E	ALR	370+17	0.019755109	0.65156386
12	5E	AL	1387+48	0.018955566	0.670519425
13	BF	ICM	MASK = C	0.0186	0.689169425
14	48	LH	DBLWRD AL	0.015491222	0.704660647
15	96	1	MASK=7C	0.014900326	0.719560974
16	94	NI	MASK=C7	0.014566785	0.734127759
17	98	LM	LM 0,10	0.014251283	0.748379041
18	7	BCR	TAKEN	0.013875519	0.76225456
19	D5	CLC	L=8&.OVLP	0.013616677	0.775871237
20	95	CLI	MASK=C7	0.011222828	0.787094065
21	90	STM	STM 0,15	0.010872599	0.797966664
22	43	IC	DBLWRD AL	0.010836294	0.808802957
23	54	N	OP1=OP2	0.010751272	0.81955423
24	55	CL	OP1=OP2	0.009797534	0.829351764
25	19	CR	CR 6,4	0.008133045	0.837484809
26	92	MVI	DBLWRD AL	0.007884764	0.845369572
27	15	CLR	3,5/R3=R5	0.007097957	0.852467529
28	59	C	OP1=OP2	0.006786195	0.859253725
29	B20A	SPKA	KEY=O	0.006460318	0.865714042
30	40	STH	DBLWRD AL	0.006204943	0.871918985
31	5	BALR	TAKEN	0.005875299	0.877794284
32	45	BAL	4,16(0,5)	0.005758106	0.88355239
33	D7	XC	L=8&.OVLP	0.005569686	0.889122076
34	B6	STCTL	STCTL O,F	0.005228056	0.894350132
35	5F	SL	1387-48	0.004980784	0.899330916
36	42	STC	DBLWRD AL	0.004953682	0.904284597
37	6	BCTR	R21S 0	0.004907376	0.909191973
38	89	SLL	8 BITS	0.004636619	0.913828592
39	5A	A	1387+48	0.00455022	0.918378812
40	8B	SLA	8 BITS	0.003862383	0.922241195
41	88	SRL	8 BITS	0.003837740	0.926078935
42	BA	CS	OP1=OP2	0.00375952	0.929838455
43	0B	BSM	TAKEN	0.003629467	0.933467922
44	49	CH	DBLWRD AL	0.003612157	0.937080078
45	AF	MC	NO RUN	0.003516373	0.940596452
46	1B	SR	370-17	0.002811749	0.943408201
47	5	BALR	R2 IS 0	0.002687654	0.946095855
48	4A	AH	1387+48	0.002319776	0.948415631
49	16	OR	3,5/R3=R5	0.001930803	0.950346434
50	4C	MH	1387*48	0.001920703	0.952267137

- RISC architectures have effectively replaced more expensive special machine architectures, the PSI virtual processor can exploit the IA technology in hosting other machine environments.

Components of Performance

- z/Architecture CPI = E + S + T
  - E-term = T-code Execution cycles per dynamic z/Architecture instruction (“end-op”)
  - S-term = T-code Storage (cache/bus) cycles per z/Architecture end-op
  - T-term = Translation cycles (Execution and Storage) per z/Architecture end-op
- T = I/R = I*M
  - I = Initial Translation cycles per static z/Architecture instruction
  - R = Instruction Re-use rate = 50-100 times
  - M = T-cache Miss rate = 1-2%
- S-term effects
  - Code expansion - linear effect on miss rate
  - Data expansion - square-root effect
  - Frequency of Translator invocations
  - Duration of Translator invocations

Translation Strategy

- Trade-off between E-, S- and T-term
  - Optimization of T-code improves E-term but worsens T-term and (indirectly) S-term
  - At T-cache miss rate of 1-2%, aggressive optimization does not pay off
- Optimization Threshold
  - Invoke optimization only after a certain instruction re-use threshold is reached
  - Performance suffers if threshold is too high or too low
  - Benefit appears marginal at best for z/Architecture
- Translation Off-load Engine
  - Requires decoupling of translation and execution so Translator can "get ahead" – difficult to achieve with z/Architecture
  - Might work with optimization threshold
- Bottom Line: Keep Translation Simple

T-Code Sample - SRA (Shift Right Arithmetic)

T-CODE TEMPLATE

```
//
// Copyright (c) 1999-2007 Platform Solutions, Inc., All Rights Reserved
// PROPRIETARY AND CONFIDENTIAL
//
//getr sraamt
SRA_B0:
{
.mii
            mov sraamt = ?D2                //shift amt gen (frc D2<64)
            sxt4 ?R1 = ?R1                  //extend sign
            nop 0x0 ;;
}
SRA_NB0:
{
.mii
            adds sraamt = ?D2,?B2           //shift amount gen
            sxt4 ?R1 = ?R1 ;;               //extend sign
            and sraamt = 63,sraamt ;;       //shift amount gen

}
//SRA_B0_CONTD:
{
.mii
            mov cctype = SZ                 //set CCTYPE
            shr ccparm1 = ?R1,sraamt        //save result as ccparm1
            shr ?R1 = ?R1,sraamt            //shift R1
}
//relr sraamt
```

T-CODE EMITTER

```
//
// Copyright (c) 1999-2007 Platform Solutions, Inc., All Rights Reserved
// PROPRIETARY AND CONFIDENTIAL
//
//390op 8a EMIT_SRA 2
EMIT_SRA:
            mov trcctyp = SZ                // set cc type
//ppmac TPARSE(1,RX)
//ppmac TSET_M(?R1,1)
   (bse20)  br.dpnt EMIT_SRA_B0             // base 0?
//ppmac TTEST_S(?B2,?R1,EMIT_SRA_NB0)
//ppmac TGET(SRA_NB0)
//ppmac TMOD(0,adds,R3,?B2)
//ppmac TMOD(0,adds,imm13,?D2)
//ppmac TMOD(1,sxt,R1,?R1)
//ppmac TMOD(1,sxt,R3,?R1)
//ppmac TPUT(f)
EMIT_SRA_B0_CONTD:
//ppmac TGET(***)
//ppmac TMOD(1,**OP**,R3,?R1)
//ppmac TMOD(2,**OP**,R1,?R1)
//ppmac TMOD(2,**OP**,R3,?R1)
//ppmac TPUT_X05()
            br.ret.sptk b2                  // return
EMIT_SRA_B0:
//ppmac TTEST_S(?R1,EMIT_SRA_B0)
//ppmac TGET(SRA_B0)
            and ?D2 = 63,?D2                // shift amt generation
//ppmac TMOD(0,mov,imm7b,?D2)
//ppmac TMOD(1,sxt,R1,?R1)
//ppmac TMOD(1,sxt,R3,?R1)
//ppmac TPUT(f)
            add xadrs = 16,xadrs            // generate bundle adrs
            br.sptk EMIT_SRA_B0_CONTD       // go emit the remainder
```

SRA T-Code Emission Trace

```
        EMIT_SRA+000001:             adds     r62 = 0x1, r62
        EMIT_SRA+000002:             br.call.sptk.few  b3 = IF_RS+000000  }
           IF_RS+000000:  {          cmp.eq   p9, p0 = 0x0, r0
           IF_RI+000001:             ld1      r107 = [r62]
           IF_RI+000002:             adds     r62 = 0x1, r62 ;;         }
           IF_RI+000010:  {          nop.m    0x0
           IF_RI+000011:    (p19=0)  or       r107 = r107, r66 ;;
           IF_RI+000012:             extr.u   r112 = r107, 0x4, 0x4     }
           IF_RI+000020:  {          and      r111 = 0x0f, r107
           IF_RI+000021:             ld1      r108 = [r62]
           IF_RI+000022:             adds     r62 = 0x1, r62 ;;         }
           IF_RI+000030:  {          ld1      r109 = [r62]
           IF_RI+000031:             adds     r62 = 0x1, r62 ;;
           IF_RI+000032:             dep      r109 = r108, r109, 0x08, 0x4  }
           IF_RI+000040:  {          nop.m    0x0
           IF_RI+000041:             extr     r108 = r108, 0x4, 0x3c
           IF_RI+000042:             cmp.eq   p36, p37 = 0x0, r111 ;;   }
           IF_RI+000050:  {          cmp.eq   p38, p39 = 0x0, r108
           IF_RI+000051:             adds     r110 = 0x20, r112
           IF_RI+000052:             adds     r113 = 0x10, r112 ;;      }
           IF_RI+000060:  { (p37=0)  adds     r111 = 0x20, r111
           IF_RI+000061:             adds     r114 = 0x10, r108
           IF_RI+000062:    (p39=0)  adds     r108 = 0x20, r108         }
           IF_RI+000070:  { (p9=1)   adds     r86 = 0x4, r86
           IF_RI+000071:    (p9=1)   adds     r81 = 0x2, r81
           IF_RI+000072:    (p9=1)   br.ret.sptk.few  b3               }
        EMIT_SRA+000010:  {          addl     r117 = 0x1, r0 ;;
        EMIT_SRA+000011:             nop.m    0x0
        EMIT_SRA+000012:             shl      r117 = r117, r110 ;;      }
        EMIT_SRA+000020:  {          or       r73 = r73, r117
        EMIT_SRA+000021:    (p38=1)  br.cond.dpnt.few
     EMIT_SRA_B0+000000:  {          nop.m    0x0
     EMIT_SRA_B0+000001:             nop.i    0x0
     EMIT_SRA_B0+000002:    (p32=1)  br.cond.dpnt.few  }
EMIT_SRA_B0_SDONE+000000:  {          addl     r102 = 0x2990, r2 ;;
EMIT_SRA_B0_SDONE+000001:             ld8      r103 = [r102], 0x08
EMIT_SRA_B0_SDONE+000002:             nop.i    0x0 ;;                    }
EMIT_SRA_B0_SDONE+000010:  {          ld8      r104 = [r102], 0x08
```

```
EMIT_SRA_B0_SDONE+000011:             and      r109 = 0x3f, r109 ;;
EMIT_SRA_B0_SDONE+000012:             dep      r103 = r109, r103, 0x12, 0x7 ;;  }
EMIT_SRA_B0_SDONE+000020:  {          nop.m    0x0
EMIT_SRA_B0_SDONE+000021:             dep      r103 = r110, r103, 0x34, 0x7
EMIT_SRA_B0_SDONE+000022:             dep      r104 = r110, r104, 0x2, 0x7 ;;  }
EMIT_SRA_B0_SDONE+000030:  {          st8      [r96] = r103, 0x08 ;;
EMIT_SRA_B0_SDONE+000031:             st8      [r96] = r104
EMIT_SRA_B0_SDONE+000032:             and      r104 = 0x70, r96 ;;       }
EMIT_SRA_B0_SDONE+000040:  {          adds     r83 = 0x1, r83
EMIT_SRA_B0_SDONE+000041:             adds     r94 = 0x1, r94
EMIT_SRA_B0_SDONE+000042:    (p33=1)  cmp.eq.unc  p40, p0 = 0x70, r104 ;;  }
EMIT_SRA_B0_SDONE+000050:  {          cmp.eq.or  p33, p0 = 0x70, r104
EMIT_SRA_B0_SDONE+000051:    (p40=0)  fc       r96
EMIT_SRA_B0_SDONE+000052:             adds     r96 = 0x08, r96           }
EMIT_SRA_B0_SDONE+000060:  {          nop.m    0x0
EMIT_SRA_B0_SDONE+000061:             adds     r102 = 0x10, r102
EMIT_SRA_B0_SDONE+000062:             br.cond.sptk.few  }
EMIT_SRA_B0_CONTD+000000:  {          ld8      r103 = [r102], 0x08 ;;
EMIT_SRA_B0_CONTD+000001:             ld8      r104 = [r102], 0x08
EMIT_SRA_B0_CONTD+000002:             nop.i    0x0 ;;                    }
EMIT_SRA_B0_CONTD+000010:  {          nop.m    0x0
EMIT_SRA_B0_CONTD+000011:             dep      r104 = r110, r104, 0x2, 0x7 ;;
EMIT_SRA_B0_CONTD+000012:             dep      r104 = r110, r104, 0x1d, 0x7 ;;  }
EMIT_SRA_B0_CONTD+000020:  {          nop.m    0x0
EMIT_SRA_B0_CONTD+000021:             nop.f    0x0
EMIT_SRA_B0_CONTD+000022:             dep      r104 = r110, r104, 0x2b, 0x7  }
EMIT_SRA_B0_CONTD+000030:  {          st8      [r96] = r103, 0x08 ;;
EMIT_SRA_B0_CONTD+000031:             st8      [r96] = r104, -0x08
EMIT_SRA_B0_CONTD+000032:             adds     r74 = 0x0, r117           }
EMIT_SRA_B0_CONTD+000040:  {          cmp.eq   p0, p32 = 0x0, r0
EMIT_SRA_B0_CONTD+000041:             adds     r83 = 0x1, r83
EMIT_SRA_B0_CONTD+000042:             adds     r94 = 0x1, r94            }
EMIT_SRA_B0_CONTD+000050:  {          nop.m    0x0
EMIT_SRA_B0_CONTD+000051:             nop.i    0x0
EMIT_SRA_B0_CONTD+000052:             br.ret.sptk.few  b2               }
```

- 27 Cycles best case
- 2.8 Instructions/Cycle best case (2.3 excluding NOPs)
- 18% NOPs – Impacts both E-term and S-term
- ~200 Cycles measured in memory-intensive CICS workload

IA64 Performance Impact

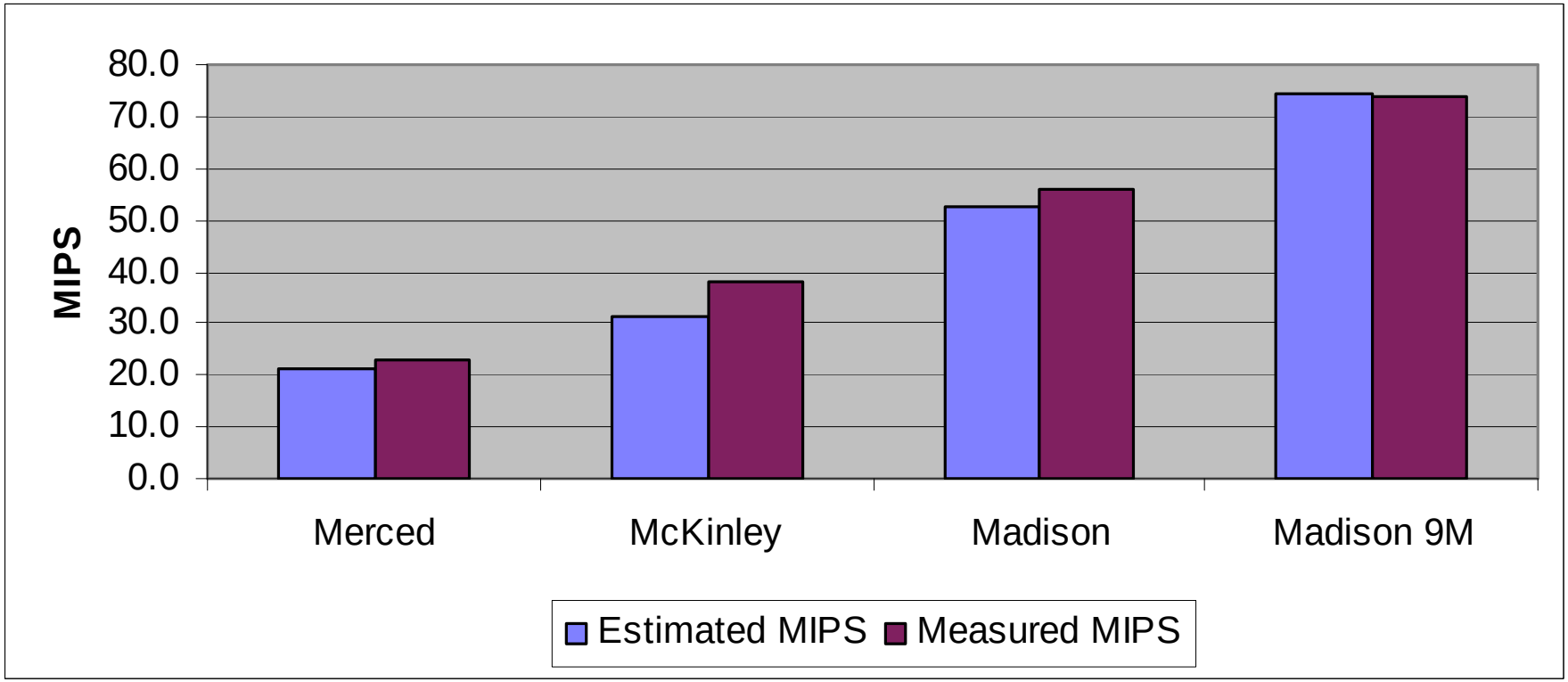
- 64-bit Architecture
  - Easier to map z/Architecture address spaces
  - Room for translator and its data structures
- 128 Registers
  - Reduced memory traffic
  - Leaner, faster code is possible
- Explicit Parallelism
  - Permits a high degree of execution overlap
  - Optimization is even more costly, however
- Code Density
  - Impacted by instruction bundle format and no-op frequency
  - Exacerbates S-term to some extent
- Good Fit Overall for z/Architecture

REV Benchmark

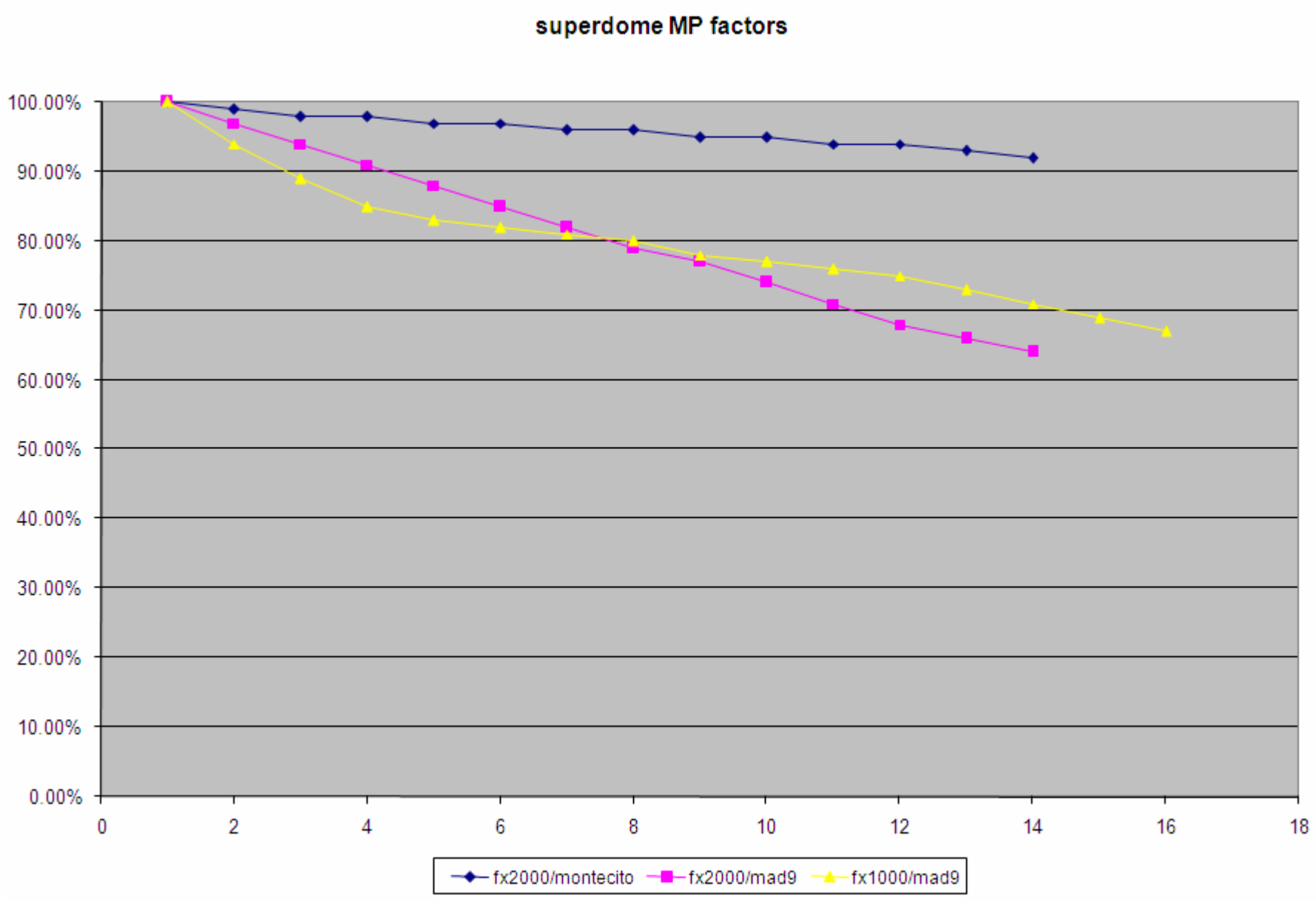
- Generated from trace of an actual MVS workload via a "reverse tracer" tool
- 1,450,407 end-ops/pass (default = 2 passes)
- Nominally accurate to +/-5%
- Calibrated over a variety of IBM/Amdahl platforms:

	REV		AMDAHL SPA		GARTNER		IBM LSPR	
MACHINE	MEAS MIPS	MIPS (*1.467)	MIPS	% DIFF	MIPS	% DIFF	MIPS (*50)	%DIFF
9672-R11	11.30	16.6	-	-	14.0	+18.6	16.5	+0.6
5890-190E	15.95	23.4	22.1	+5.9	22.4	+4.5	23.5	-0.4
3090-180J	17.86	26.2	25.4	+3.1	23.5	+11.5	25.5	+2.7
5990-350	21.56	31.6	32.9	-4.0	32.0	-1.3	32.0	-1.3
5995M (2X)	40.91	60.0	60.0	0.0	55.0	+9.0	55.5	+8.1
5995M (8X)	45.07	66.1	67.0	-1.3	-	-	-	-
MILL GS5xx	33.94	49.8	49.0	+1.6	47.0	+6.0	-	-
MILL GS7x4	57.46	84.3	80.0	+5.4	76.0	+10.9	-	-
MILL GS8x4	61.26	89.9	98.0	-8.3	-	-	-	-

PSI Single Processor Performance



PSI System Performance



Future Directions

- Keep pace with IPF enhancements
  - Continually re-optimize T-code for future IPF processors and servers
  - Enable customer exploitation of hyperthreading
  - Utilize VT for more flexible/granular virtualization/partitioning
- Pre-processor and Development Environment Improvements
  - Table-driven T-code and emitter generation
  - Utilize RSE for deeply nested routines
  - Utilize C for infrequently called routines
- Performance Improvement
  - SAP-based “hot code” optimization
  - “L2” T-cache (i.e. saved/optimized T-code to reduce re-translation)
- Improve Legacy/Open Interoperability
  - In-Memory TCP/IP between z/OS and Open Systems
  - Native Itanium-based off-load processors
  - Legacy architectural extensions

Call to Action

- IA-64 Processor Architecture
  - Make it more JIT-friendly
    - Optimize fc.i providing both local and global versions
    - Consider implementing instruction cache snoop/coherency
    - Relax instruction dispersal restrictions (i.e. simplify scheduling)
    - Consider adding “double-comma” bundles for improved code density
  - Continue performance improvement
    - Larger, faster caches – especially instruction cache
    - GHz still matters – single-threaded CPU-intensive workloads
    - Multi-core – enterprise appetite for capacity is huge and growing
  - Maintain/enhance legacy support
    - 4K page size is critical
    - Efficient strong memory ordering – consider adding st.mf
    - Consider adding cmpxchg16
  - RAS – five 9’s minimum
- EPIC Compiler Technology
  - Highly efficient stand-alone back-end optimizer for JIT run-time use
  - Optimize for code size reduction even more than cycle reduction
  - Bit manipulation utilizing IPF primitives (e.g. extr/dep) where possible
  - A fun challenge – match/beat SRA T-code/emitter efficiency in C!